aerospace

MDPI

Article

Efficient Inference of Neural Networks with Cooperative Integer-Only Arithmetic on a SoC FPGA for Onboard LEO Satellite Network Routing

Bogeun Jo [1], Heoncheol Lee [1,*], Bongsoo Roh [2] and Myonghun Han [2]

[1] Department of IT Convergence Engineering, Kumoh National Institute of Technology, Gyeongbuk 39177, Republic of Korea; qhrmsgo@kumoh.ac.kr
[2] Agency for Defense Development, Daejeon 34186, Republic of Korea; saintroh@add.re.kr (B.R.); mengddor@add.re.kr (M.H.)
* Correspondence: hclee@kumoh.ac.kr; Tel.: +82-54-478-7476

Abstract

Low Earth orbit (LEO) satellite networks require real-time routing to cope with dynamic topology variations caused by continuous orbital motion. As an alternative to conventional routing approaches, deep reinforcement learning (DRL) has recently gained attention as an effective means for optimizing routing paths. To solve routing problems modeled as a grid-based Markov decision process (grid-based MDP), DRL methods such as CNN-based Dueling DQN have been proposed. However, these approaches are difficult to implement in practice. In particular, the substantial floating-point computation and memory traffic of CNN inference make real-time onboard inference challenging under the stringent power and resource constraints of satellite platforms. To address these constraints, this paper proposes an INT8 quantization and hardware–software co-design framework using heterogeneous SoC FPGA acceleration. We offload compute-intensive CNN inference to the programmable logic (PL), while the processing system (PS) orchestrates overall control and data movement, forming a collaborative PS–PL architecture. Furthermore, we integrate the NITI-style two-pass scaling with PS–PL exponent propagation to preserve end-to-end integer consistency without floating-point conversion. To demonstrate its practical onboard feasibility, we employ standard accelerator implementation choices—such as output-stationary scheduling and on-chip prefetching—and conduct an ablation study over independently tunable axes (PE array size and PS-side buffer reuse) to quantify their incremental contributions. Experimental results show that the proposed PS–PL cooperative scheme dramatically reduces computation time compared to a PS-only reference implementation on the same platform.

Keywords: LEO satellite networks; onboard routing; heterogeneous SoC FPGA; deep reinforcement learning; CNN acceleration; onboard computer; INT8 quantization

Academic Editor: M. Reza Emami

Received: 6 February 2026
Revised: 10 March 2026
Accepted: 13 March 2026
Published: 16 March 2026

1. Introduction

Low Earth orbit (LEO) satellite networks have recently attracted active research interest because, by leveraging mega-constellations comprising hundreds or more satellites, they can provide wide-area coverage and low-latency communications even in regions with weak terrestrial infrastructure [1–3]. However, because satellites orbit the Earth at high speed, link connectivity and network topology change rapidly over time, leading to frequent link disruptions and recoveries [1,4,5]. These dynamic characteristics make

it unrealistic to assume static routing path planning. Therefore, routing in LEO satellite networks entails a frequent decision-making problem in which paths must be computed in real time while accounting for periodically varying topology as well as link disruptions and recoveries [4,5].

To address this problem, a number of dedicated routing algorithms have been proposed that reflect the dynamic environment and delay characteristics of satellite networks. For example, a distributed routing algorithm for datagram traffic in LEO networks has been proposed [6], and MLSR (Multi-Layered Satellite Routing), which efficiently updates routing tables based on delay measurements for multi-layer (LEO/MEO/GEO, etc.) satellite IP networks, has also been presented [7]. Approaches have also been discussed that mitigate topology variations by defining "virtual topologies" for specific time intervals, exploiting the fact that satellite orbital motion is relatively predictable [8]. Nevertheless, as network scale increases and link-state variability grows, it becomes difficult to guarantee consistent performance across diverse operating conditions, and such approaches have limitations in adaptive decision-making that can flexibly respond to complex environmental changes [5].

Accordingly, research leveraging reinforcement learning and deep reinforcement learning (DRL) to adapt to complex network environments has been increasing. Reinforcement-learning-based routing enables an agent to learn a policy that selects optimal paths through interaction with the environment, offering the advantage of rapid adaptation to changing satellite network states [9]. In particular, techniques that represent satellite networks in grid or matrix-form states and process them with convolutional neural networks (CNNs) have gained attention, as they effectively extract spatial characteristics such as topology dynamics and traffic distributions [10–13]. These CNN-based DRL models have demonstrated the potential to establish robust routing strategies even in large-scale constellation environments [5].

However, there remain critical bottlenecks in applying DRL-based routing algorithms to practical onboard satellite environments. In LEO settings, routing decisions must be recomputed frequently, and accumulated inference latency makes it difficult to respond promptly to fast environmental changes such as link disruptions and recoveries. In particular, in CNN-based DRL models, convolution operations for feature extraction demand substantial computation, and conventional floating-point (fp32) implementations increase both computational cost and memory usage, making real-time deployment difficult under the strict power, thermal, and resource constraints of onboard computing, as illustrated in Figure 1 [14,15]. In other words, to make DRL practical for LEO onboard routing, a method is required that simultaneously reduces inference computation and data-movement overhead under limited resources while preserving accuracy [14,16,17].

To address these constraints, this paper proposes an onboard CNN inference acceleration framework that combines an integer-only numerical scheme with a SoC FPGA–based hardware–software co-design. Specifically, assuming an NITI-based INT8 model that consistently uses integer arithmetic starting from training [17], we construct an inference pipeline that operates exclusively with integer arithmetic at inference time as well, without any floating-point conversion [16,17]. In addition, by exploiting the heterogeneous architecture of an SoC FPGA, computation-intensive convolution operations are offloaded to the PL (Programmable Logic), while the PS (Processing System) handles layer execution control and data movement as well as non-convolution operations, thereby forming a cooperative PS–PL inference architecture.

We define this cooperation as a dual mechanism: (1) architectural cooperation, where the PS manages control and the PL executes compute-intensive kernels; and (2) numerical cooperation, where scaling metadata is exchanged to ensure end-to-end integer-only arithmetic.

This work does not claim that the underlying accelerator primitives are exclusive to LEO routing. Rather, it targets the deployment bottleneck of the CNN-based feature

extractor inside a Dueling DQN routing agent for dynamic LEO satellite networks, where strict onboard power, thermal, and resource constraints make end-to-end integer-only inference a practically relevant system problem.

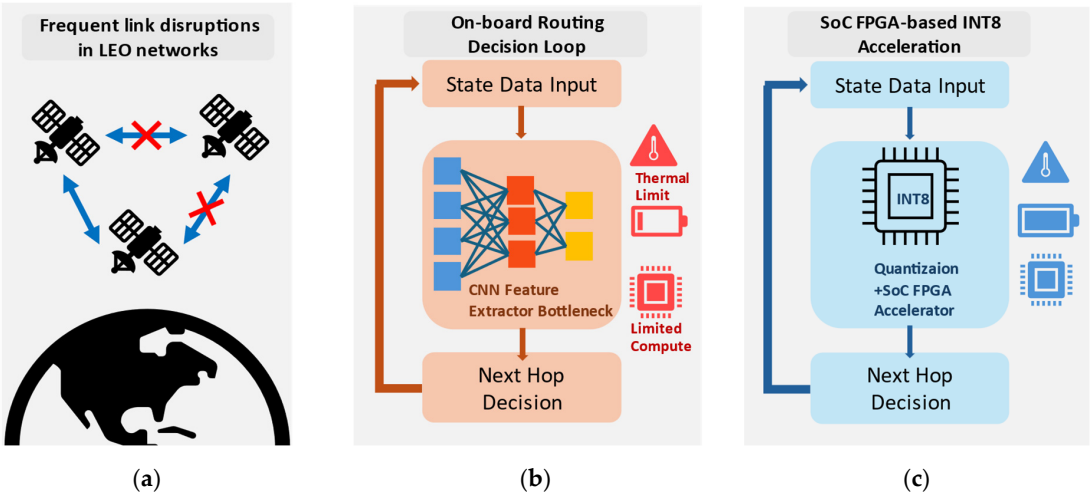

Figure 1. Motivation of this work. (**a**) In LEO satellite constellations, high-speed orbital motion creates a dynamic network topology, which may cause frequent disruptions in Inter-Satellite Links. (**b**) Such instability necessitates real-time onboard routing decisions; however, the heavy computational overhead of CNN-based models conflicts with the strict power, thermal, and computing limits of satellite platforms. (**c**) These onboard constraints render standard floating-point CNN inference impractical, motivating the proposed INT8 integer-only architecture and PS–PL SoC FPGA acceleration framework.

The contributions of this paper are as follows.

1. Implementation framework for NITI-based INT8 CNNs: Unlike conventional fixed-point FPGA accelerators, this work presents a PS–PL cooperative architecture specifically designed to support the dynamic scaling rules of the NITI framework. We bridge the gap between integer-only training theory and hardware deployment by implementing a two-pass mechanism that maintains numerical consistency without any floating-point units.
2. Cooperative Integer-only Arithmetic architecture: We propose a heterogeneous cooperative architecture in which the PL performs convolution and integer accumulation, computes the information required for the integer scaling (shift/scale exponent) of outputs, and delivers it to the PS, while the PS uses it for inter-layer scale propagation and overall execution control.

Ablation analysis of the proposed PS–PL framework: We analyze the performance contributions of key design variables within our cooperative pipeline. A 2×2 ablation study is conducted on the PE array size and buffer reuse logic, providing a quantitative breakdown of the speedup achieved by our integration of integer-only arithmetic and SoC FPGA acceleration. This study focuses on the CNN-based feature extractor, which accounts for a substantial computational burden within the overall routing pipeline. In addition, to reproducibly validate the hardware feasibility and performance of the proposed SoC FPGA–based integer-only inference structure, we implement and evaluate, on an embedded platform (ZCU104), an image classification task (CIFAR-10 [18], VGG-SMALL-7 [13] trained weights) using a NITI-based INT8 image-classification model (CIFAR-10 with VGG-SMALL-7 weights) as a proxy workload for the convolution-dominant inference path [17]. The proxy preserves implementation-relevant characteristics of the target pipeline—including repeated 3×3 convolution execution, integer scale propagation,

and repeated layer invocation—but is not a scale-matched surrogate of the routing CNN. Accordingly, the reported results should be interpreted as a hardware-feasibility demonstration of the proposed PS–PL inference pipeline rather than as direct performance evidence for the routing workload itself. In this work, the software-side comparison is made against a PS-only software reference executed on the same SoC platform. This reference preserves the same integer arithmetic rules as the proposed pipeline but is not intended to represent a fully optimized ARM INT8 inference engine. The remainder of this paper is organized as follows. Section 2 formulates the LEO network routing problem from a reinforcement learning perspective and describes the CNN-based Dueling DQN architecture and the computational bottlenecks arising in onboard deployment. Section 3 presents the SoC FPGA–based cooperative PS–PL integer-only inference structure and the three optimization strategies. Section 4 analyzes inference latency/throughput and resource utilization through experiments on the ZCU104 evaluation board (AMD, Santa Clara, CA, USA), and Section 5 concludes the paper and discusses future research directions.

2. Problem Description

2.1. Grid MDP Formulation for LEO Satellite Routing

A low Earth orbit (LEO) satellite network can be modeled as a time-varying dynamic graph whose topology changes over time due to the high-speed motion of satellites [1,4,5]. In such an environment, routing is defined as a sequential decision-making problem in which the next hop is repeatedly selected to deliver a packet from the current node to the destination. To address this routing decision process from a reinforcement learning perspective, this study formulates it as a Markov Decision Process (MDP) [9], and, in particular, adopts a Grid MDP, in which the state space is constructed in a grid form to reflect the spatial structure of the satellite constellation.

The Grid MDP is formulated as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$.

- State ($\mathcal{S}$): To enable the CNN to learn spatial correlations, we represent the state s_t as a binary tensor of dimension $4 \times H \times W$. The state s_t at time t is constructed by stacking the following four channels. Channel 1 (M_{agent}) and Channel 2 (M_{target}) are one-hot matrices indicating the locations of the current agent (satellite) and the destination satellite of the packet, respectively. Channel 3 (M_{bound}) is a mask encoding the boundary information of the grid map, where valid regions are labeled as 1 and padded regions as 0, allowing the agent to recognize the map extent. Channel 4 ($M_{obs}(t)$) is a matrix representing the distribution of dynamic obstacles whose locations vary with time t (e.g., link-disruption segments, congested satellites).

$$s_t = \left\{ M_{agent}, M_{target}, M_{bound}, M_{obs}(t) \right\}$$

- Action ($\mathcal{A}$): The action $a_t \in \mathcal{A}$ is defined as one of five discrete moves available to the agent in the grid environment:

$$\mathcal{A} = \{\text{UP, DOWN, LEFT, RIGHT, WAIT}\}$$

- Transition Probability (P): While the agent's movement is deterministic, the obstacle map $M_{obs}(t)$ changes periodically according to the environment's temporal pattern. Accordingly, the next state s_{t+1} is determined by combining the agent's movement outcome with the updated obstacle map.
- Reward (R): The reward function r_t is designed to facilitate learning of an optimal path as follows. A positive reward (R_{end}) is given upon reaching the destination, whereas penalties (negative rewards) of different magnitudes are assigned for obstacle

collisions (R_{obs}), staying in place (R_{stop}), simple waiting (R_{wait}) and regular movement (R_{step}), thereby encouraging collision-free shortest-path search.

2.2. CNN-Based Dueling DQN Architecture

To derive an optimal policy in a high-dimensional grid state space, this study employs the Dueling DQN (Deep Q-Network) algorithm [10]. Dueling DQN estimates the Q-function by decomposing it into a state-value term and an advantage term. Here, $V(s)$ denotes the value of the current state, and $A(s, a)$ represents the relative advantage of taking a particular action in that state; these are combined to form $Q(s, a)$. A representative aggregation is given by [11]:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

Since the state s_t has an image-like tensor structure of size $4 \times 15 \times 15$, we use a 2D convolutional neural network (2D-CNN) as a feature extractor to effectively capture spatial features [12].

Given the compact grid-state input, we employ a lightweight CNN feature extractor for the Dueling DQN routing agent. In our target routing model, the feature extractor consists of four sequential 3×3 convolution layers with channel widths {16, 32, 128, 128}, each followed by a ReLU activation, and no pooling is applied to avoid excessive spatial downsampling on the small grid input. The final feature map ($128 \times 7 \times 7$) is flattened and then processed by fully connected layers before branching into the value and advantage heads of the Dueling DQN. In this architecture, the multi-layer convolution operations constitute the primary bottleneck, accounting for the majority of total inference time and memory traffic. The grid state for routing (4-channel) shares the same data structure and computational pattern as standard image datasets (e.g., 3-channel RGB). Therefore, to ensure objective and reproducible validation of the proposed accelerator architecture without introducing unverified reinforcement learning policy variations, we generalize the dataset channel configuration and primarily evaluate the acceleration performance of the VGG-based feature extractor under a standard benchmark (CIFAR-10) setting as a proxy workload [13,18]. To make the relationship between the proxy and the target routing feature extractor explicit, Table 1 summarizes the layer-wise tensor shapes, and Table 2 reports the corresponding compute and memory footprints.

Table 1. Layer-wise Tensor Shape Comparison (Routing DuelingDQN vs. Proxy VGG-SMALL-7): Routing conv: 3×3, stride = 1, padding = 0 (valid). Proxy conv: 3×3, stride = 1, padding = 1 (same spatial), with 2×2 max pooling (stride = 2).

Stage	Routing CNN (DuelingDQN) In (Cin × H × W)	Routing CNN Out (Cout × H × W)	Proxy CNN In (VGG-SMALL-7)	Proxy CNN Out (Cout × H × W)
Input	4 × 15 × 15	4 × 15 × 15	3 × 32 × 32	3 × 32 × 32
Conv1	4 × 15 × 15	16 × 13 × 13	3 × 32 × 32	128 × 32 × 32
Conv2	16 × 13 × 13	32 × 11 × 11	128 × 32 × 32	128 × 32 × 32
MaxPool1	None (No pooling)	—	128 × 32 × 32	128 × 16 × 16
Conv3	32 × 11 × 11	128 × 9 × 9	128 × 16 × 16	256 × 16 × 16
Conv4	128 × 9 × 9	128 × 7 × 7	256 × 16 × 16	256 × 16 × 16
MaxPool2	None (No pooling)	—	256 × 16 × 16	256 × 8 × 8
Conv5	N/A	—	256 × 8 × 8	512 × 8 × 8
Conv6	N/A	—	512 × 8 × 8	512 × 8 × 8
MaxPool3	N/A	—	512 × 8 × 8	512 × 4 × 4
Flatten	128 × 7 × 7	6272	512 × 4 × 4	8192

Table 2. Compute and Memory Footprint Comparison (Conv layers only).

Metric	Routing CNN (DuelingDQN)	Proxy CNN (VGG-SMALL-7)	Proxy/Routing
Total Conv MACs	10,866,240 MACs	607,518,720 MACs	55.9×
Total Conv Weights (INT8, Bytes)	189,504 Bytes	4,574,592 Bytes	24.1×

As shown in Tables 1 and 2, the proxy CNN is substantially larger than the routing CNN in both MAC count and parameter size, and differs in pooling/padding configuration. These structural differences can lead to different memory-access and execution behavior on the accelerator. Accordingly, the reported results are interpreted as a hardware-feasibility demonstration of the proposed PS–PL inference pipeline rather than as direct performance evidence for the routing workload.

2.3. Onboard Constraints and Motivation for Integer-Only Inference

The onboard computing environment of LEO satellites is constrained in terms of power, thermal dissipation, and hardware resources, while still requiring repetitive and continuous routing decisions to cope with frequent topology changes and link disruptions. Under these conditions, Dueling DQN–based routing is advantageous in terms of environmental adaptability; however, it incurs the burden that high-compute CNN processing is essential for feature extraction. A software implementation solely on a general-purpose processor (CPU) is limited in computational capability for handling such complex CNN operations in real time, and excessive inference latency may result in routing failures. To address this limitation, recent prior studies have proposed leveraging embedded GPUs or employing CNN parallelization/acceleration via CPU–FPGA heterogeneous architectures, demonstrating that hardware acceleration can achieve significant performance improvements and better power efficiency compared to CPU-only execution [14,15,19].

Among these studies, ref. [19] is the closest prior work because it also addresses accelerated onboard inference for Dueling DQN-based routing in dynamic LEO satellite networks. However, the two studies address different design points. Ref. [19] focuses on GPU-based acceleration of the routing pipeline, whereas this work focuses on end-to-end integer-only inference on a heterogeneous SoC FPGA. Ref. [19] emphasizes routing-oriented evaluation, whereas our validation is positioned as a reproducible hardware-feasibility study of the convolution-dominant feature-extraction path. Accordingly, our contribution is not the use of PL or INT8 per se, but the way we integrate NITI-style two-pass scaling with PS–PL exponent propagation. For this reason, the two studies are better viewed as complementary, rather than as directly comparable benchmarks.

The core operation of CNN inference is large-scale MAC (multiply–accumulate) computation between convolution kernels and input feature maps. From a hardware implementation perspective, floating-point (e.g., FP16, bfloat16, FP32) arithmetic units are substantially more complex than integer (INT8) units, which inevitably leads to higher utilization of hardware resources (LUTs, DSPs) and greater power consumption [16,17]. NITI [17] established that low-bit integer arithmetic offers significant advantages for hardware implementation, demonstrating that FPGA/ASIC resource consumption for the same MAC operation can vary considerably depending on the data type. Consequently, floating-point–based CNN inference in onboard environments is inefficient for real-time processing not only due to high computational complexity but also due to increased memory bandwidth pressure caused by wider data bit-widths.

Therefore, to maximize hardware efficiency, this paper utilizes a CNN model trained under the NITI framework [17]. NITI is designed to use only standard 8-bit integer arithmetic

from the training stage, performing storage and updates of parameters and intermediate accumulations primarily in the INT8 format, while applying hardware-friendly operations such as per-layer scaling and shift-and-round. Unlike simple post-training quantization (PTQ), these characteristics ensure that the model's numerical representation is consistent with an integer arithmetic pipeline [16,17]. Accordingly, in Section 3, we propose a cooperative accelerator architecture on an SoC FPGA to efficiently process this Integer-Only Arithmetic.

3. Proposed Method

3.1. Key Idea: Cooperative Integer-Only Arithmetic on a SoC FPGA

To simultaneously address the inference bottleneck of onboard DRL routing formulated in Section 2 (i.e., repeated convolution operations in the feature-extractor CNN) and the stringent constraints of the satellite onboard environment in terms of limited power, resources, and bandwidth, this paper proposes a PS–PL cooperative inference architecture based on Cooperative Integer-Only Arithmetic. In this study, 'cooperative' is strictly defined beyond simple hardware offloading. It encompasses a task-partitioning scheme (PL for convolutions, PS for orchestration) and a numerical scaling protocol where the PL computes and the PS propagates integer-scale exponents. This distinguishes our approach from conventional accelerators that often rely on floating-point conversions at the PS–PL boundary.

Specifically, the PL (Programmable Logic) performs computation-intensive 3×3 convolutions with INT8 $\times$ INT8 $\rightarrow$ INT32 accumulation. Crucially, it calculates the shift values and the output scale exponent $s_{a,out}$ required to preserve the dynamic range of layer outputs—directly in hardware and returns them to the PS. The PS (Processing System) handles layer execution control (register configuration and synchronization), buffer and data movement management, and operations with relatively higher control overhead (e.g., Pooling, FC) in the integer domain without floating-point conversion. By propagating the $s_{a,out}$ returned by the PL as the input scale for the subsequent layer, the PS maintains end-to-end integer numerical consistency. Algorithm 1 Pseudo-code representing the PS–PL Cooperative Integer-Only Inference Protocol.

Algorithm 1. PS–PL Cooperative Integer-Only Inference Protocol

Input	INT8 activation $a^{(0)}$
	Input scale exponent $s_a^{(0)}$
Output	Final Class Decision
1.	**for each conv layer** ℓ **do**
2.	**PS:** Configure AXI4-Lite registers:
3.	Buffer Addrs $(\boldsymbol{a}^{(\mathbf{l})}, \boldsymbol{w}^{(\mathbf{l})}, \boldsymbol{b}^{(\mathbf{l})}, \boldsymbol{u}^{(\mathbf{l})}, \boldsymbol{a}^{(\mathbf{l}+1)})$
4.	Params $(H, W, C_{\text{in}}, C_{\text{out}}, s_a^{(l)}, s_w^{(l)})$
5.	**PL (Pass 1):** Compute INT32 Accumulation $u^{(l)}$
6.	Estimate dynamic range stats (b_{max})
7.	**PL (Pass 2):** Determine shift value
8.	Generate INT8 output $a^{(l+1)}$ via RN and Saturation
9.	**PL** $\rightarrow$ **PS:** Return output scale exponent $s_a^{(l+1)}$
10.	**PS:** Perform ReLU/Pooling (if applicable) in INT8 domain
11.	Update input for next layer $a^{(l)} \leftarrow a^{(l+1)}$
12.	**end for**
13.	**PS:** Execute FC/Head using INT8/INT32 rules
14.	**return** argmax(**Final Logits**)

3.2. Structure of a SoC FPGA

This study is implemented on a SoC FPGA (ZCU104) based on the AMD Xilinx Zynq UltraScale + MPSoC (XCZU7EV, AMD, Santa Clara, CA, USA). The PS is the general-purpose processor domain where an OS and a Python runtime execute, whereas the PL is a reconfigurable logic domain in which an HLS-based convolution accelerator IP is instantiated using DSP/LUT/BRAM resources. The PS–PL control path uses AXI4-Lite, and the data path uses AXI4 Memory-Mapped. The interface configuration of the HLS convolution IP, including the AXI4-Lite control path and AXI4 memory-mapped data ports, is summarized in Table 3.

Table 3. Summary of Interfaces for the HLS Convolution IP (niti_conv2d_hls).

Category	Port Name	Data Type	Interface	Description
Data Input	in_r	Int8	AXI4-MM (gmem0)	Input Feature Map (C_{in}, H, W)
	weights	Int8	AXI4-MM (gmem1)	Kernels $(C_{out}, C_{in}, 3, 3)$
	bias	Int32	AXI4-MM (gmem2)	Biases (C_{out})
Data Output	sum_buf	Int32	AXI4-MM (gmem4)	Pass 1 Accumulation $(C_{out}, H_{out}, W_{out})$
	out_r	Int8	AXI4-MM (gmem3)	Pass 2 Final Output $(C_{out}, H_{out}, W_{out})$
Control	H, W, C_in, C_out	Int	AXI4-Lite	Tensor Dimensions
	stride, pad	Int	AXI4-Lite	Convolution Hyperparameters
	s_a_in, s_w	Int	AXI4-Lite	Input/Weight Scale Exponents
	mode	Int	AXI4-Lite	0: All, 1: Acc Only, 2: Quant Only
Return	s_a_out	Int	AXI4-Lite	Output Scale Exponent (to PS)

In particular, this implementation separates the input/weight/bias/output/accumulation buffers into mutually independent AXI master channels (gmem0–gmem4), thereby mitigating the concentration of read/write traffic onto a single channel during the two-pass operation and ensuring parallelism in DDR accesses.

At runtime, the PS allocates physically contiguous cacheable buffers for inputs, weights, bias, outputs, and sum_buf using the pynq.allocate() API, writing their physical addresses to the AXI4-Lite control registers. Completion is detected by polling the ap_done bit rather than by interrupts. To ensure PS–PL data coherence for cacheable buffers, the software flushes input-side buffers before kernel launch and invalidates PL-produced buffers after completion.

3.3. Cooperative Integer-Only Arithmetic

To eliminate floating-point operations, this study adopts Block Exponentiation Scaling from the NITI framework. This scheme stores tensor values in INT8, while representing their dynamic range via a layer-wise shared scale exponent (exponent, s). Given an integer tensor $x \in \mathbb{Z}_8$ and a scale exponent $s_x \in \mathbb{Z}$, the corresponding value in the real domain $\hat{x}$ is defined as

$$\hat{x} = x \cdot 2^{s_x}$$

For convolution, given the input activation $a^{(l-1)}$ and the weight $w^{(l)}$, the ideal output scale s_{ideal} is simplified into exponent addition by the multiplicative property:

$$s_{ideal}^{(l)} = s_a^{(l-1)} + s_w^{(l)}$$

When scaling is restricted to powers of two, the scale adjustment required at inference time can be implemented using bit-shift–based operations rather than general multiplications, thereby reducing hardware cost.

Since convolution consists of MAC (Multiply–Accumulate) operations across many inputs and weights, intermediate accumulations exceed the INT8 range. Therefore, in this implementation, accumulation is performed in INT32 to prevent overflow:

$$y_{32}^{(l)} = \sum_k w_{int8}^{(l)}[k] \cdot a_{int8}^{(l-1)}[k] + b_{32}^{(l)}$$

When re-quantizing the INT32 result to INT8 for the next layer, we determine an optimal shift amount based on the effective bitwidth B to minimize information loss. Specifically, we compute the maximum effective bitwidth b_{max} by finding the maximum absolute value over the entire output tensor y_{32}, and then derive the shift amount based on the magnitude-bit budget, explicitly excluding the sign bit:

$$b_{max} = B(y\_\{32\}) = \lceil \log \_2(\max|y\_\{32\}| + 1) \rceil$$

$$shift = \max(0, b_{max} - 7)$$

The final output scale exponent $s_{a,out}$ is updated by incorporating the shift as follows. This value is managed by the PS based on the shift information computed in the PL, and is propagated as the input scale of the next layer to ensure system-wide numerical consistency:

$$s_a^{(l)} = s_{ideal}^{(l)} - shift = s_a^{(l-1)} + s_w^{(l)} - shift$$

The proposed IP computes b_{max} in the PL to determine the shift and returns the resulting $s_{a,out}$ to the PS via AXI4-Lite, thereby enabling inter-layer scale propagation [17].

During inference, Round-to-Nearest (RN) is used for reproducibility and hardware simplicity. Given a shift value, RN-based shifting can be defined as.

$$\widetilde{y} = Round\left(y_{32} \cdot 2^{-shift}\right)$$

$$y_{int8} = Sat_{[-128,127]}(\widetilde{y})$$

To ensure numerical consistency between the PS and PL domains, the signed requantization and Round-to-Nearest (RN) rule are formally defined. Given an INT32 accumulation result x and a right-shift amount s:

$$RN(x,s) = \begin{cases} x, & s = 0 \\ sign(x)\left\lfloor \frac{|x|+2^{s-1}}{2^s} \right\rfloor, & s > 0 \end{cases}$$

$Sat(\cdot)$ denotes saturation clipping to the explicitly signed INT8 bounds:

$$y_{int8} = clip(RN(x,s), -128, 127)$$

Dynamic scaling has a causal constraint: the shift cannot be determined until b_{max} over the entire output tensor is finalized. In a streaming fashion, the maximum bitwidth of "outputs not yet observed" is unknown; thus, to faithfully reproduce the NITI rules, this

study employs a Two-Pass mechanism. In Pass 1 (Statistics Collection), the PL accelerator performs convolution, generates the INT32 result (y_{32}) and stores it in a buffer (sum_buf), while the hardware simultaneously tracks and updates the maximum output bitwidth (b_{max}) in real time. In Pass 2 (Quantization and Output), the shift value is computed based on the finalized b_{max}; the buffered y_{32} is then read back to perform Shift-and-Round and Saturation, producing the final INT8 output. Although the two-pass scheme incurs additional DDR traffic to the accumulation buffer (sum_buf), it guarantees numerical consistency of integer-only inference by accurately reproducing the per-layer dynamic-range–aware scaling rule in hardware.

To explicitly quantify the overhead of the proposed two-pass mechanism, we added a profiling-only execution mode (Mode 2) that performs the same first-pass accumulation and b_{max} extraction as the normal first pass, but suppresses the full INT32 intermediate-buffer materialization. A single checksum write is retained only to prevent logic elimination during synthesis. Using this mode, the two-pass overhead can be decomposed into the latency contribution of INT32 intermediate-buffer materialization and the latency contribution of the second sweep for requantization and output generation.

For a convolution layer with an output tensor size of $C_{out} \times H_{out} \times W_{out}$, the exact additional external-memory traffic introduced by the two-pass design is given by:

$$\Delta B = 8 \cdot C_{out} \cdot H_{out} \cdot W_{out} \text{ bytes}$$

This is because each output element requires one INT32 write (4 bytes) and one INT32 read (4 bytes) of the intermediate sum buffer.

3.4. SoC FPGA-Based Inference Parallelization Method #1: Output-Stationary Tiled MAC Array Parallelism

The first parallelization strategy maximizes operation-level parallelism in the PL-side convolution kernel by adopting an output-stationary dataflow that keeps accumulators fixed in registers with respect to an output tile, and by fully parallelizing the output-channel and output-width dimensions within each tile.

A 3×3 convolution is defined over the output channel (OC) and output spatial dimensions (OH, OW) as

$$y_{[oc,oh,ow]} = \sum_{ic} \sum_{ky} \sum_{kx} w_{[oc,ic,ky,kx]} \cdot a_{[ic,oh \cdot s - p + ky, ow \cdot s - p + kx]} + b_{[oc]}$$

This design reinterprets the operation from a matrix-multiplication perspective: the output-channel dimension (M) and the output-width dimension (N) are tiled with sizes SA_M and SA_N, respectively, as shown in Figure 2, while the K axis—comprising the input channels and kernel window—is used as the accumulation axis. In the implemented design, the output-channel tile size SA_M and output-width tile size SA_N are fixed to 16 and 16, respectively, yielding a 16×16 output-stationary compute tile. The inner K-loop is explicitly pipelined with a target Initiation Interval (II) of 1 (the actual achieved II is 1).

For an output tile, the accumulators $acc[m][n]$ are kept in a register array within the PL. By fully unrolling the loops over $m \in [0, M)$ and $n \in [0, N)$, we construct an $M \times N$ parallel MAC array. The reduction loop along the accumulation axis (K) is pipelined to improve throughput under continuous data supply, and correctness is ensured by applying conditional checks at tile boundaries so that only valid operations are executed.

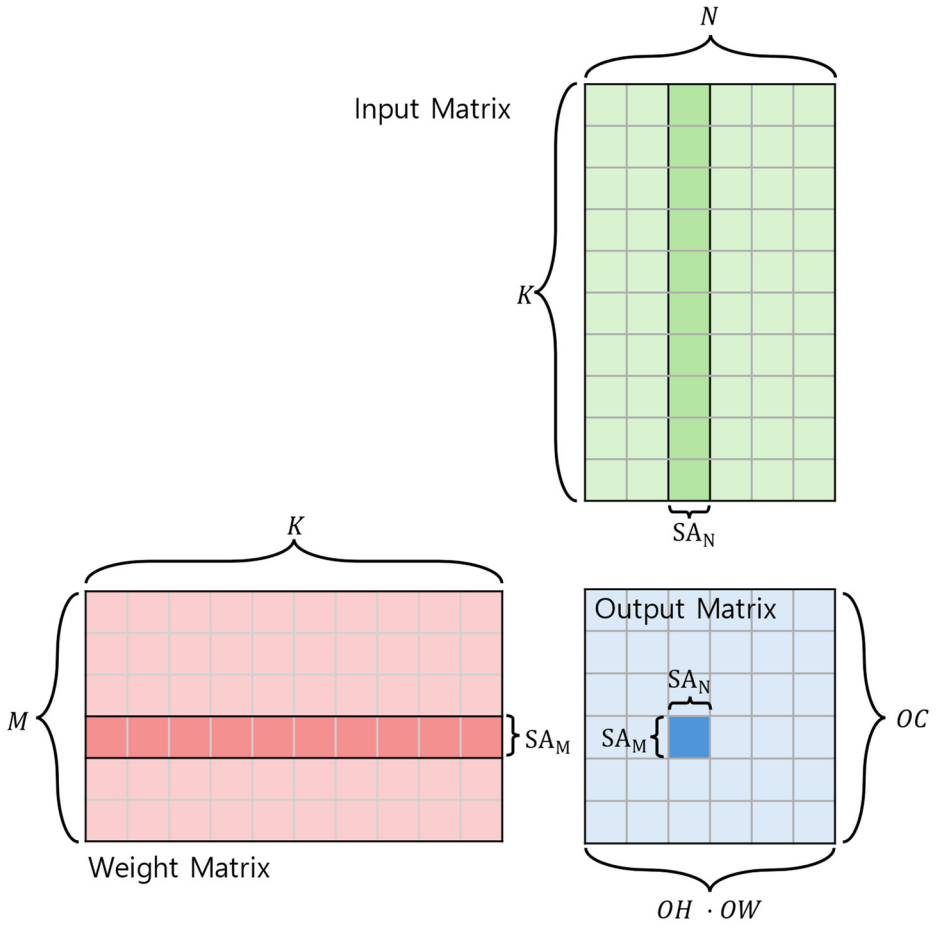

Figure 2. Tiled matrix multiplication strategy for convolution acceleration.

3.5. SoC FPGA-Based Inference Parallelization Method #2: On-Chip Prefetching and Pipelining

The second strategy aims to minimize access to external memory (DDR), which is the primary bottleneck of the convolution accelerator, by promoting tile-level data reuse to on-chip BRAM and eliminating DDR accesses from the compute-intensive accumulation loop.

The proposed IP uses the following on-chip tile buffers:

- Weight tile: $w_{\text{tile}} \in Z_{\mathbb{8}}^{M \times K}$
- Activation tile: $a_{\text{tile}} \in Z_{\mathbb{8}}^{N \times K}$

For each output tile, we prefetch w_{tile} and a_{tile} from DDR, and then in the innermost K-loop, update $acc[m][n]$ by reading only from the tile buffers in BRAM. This removes DDR reads from the accumulation loop, thereby alleviating the memory bottleneck. The accumulation loop is also designed to minimize the pipeline initiation interval (II).

To support multiple concurrent accesses, the BRAM-based tile buffers are fully partitioned along relevant dimensions and bound to dual-port BRAMs to mitigate port conflicts. At the system level, as depicted in Figure 3, the input/weight/bias/output/ accumulation buffer (sum_buf) interfaces are separated into independent AXI bundles so that the read/write traffic generated during the two-pass procedure does not concentrate on a single channel.

To support NITI's dynamic scaling, the hardware pipeline is organized into two passes. Pass 1 performs convolution to produce INT32 accumulation results and searches, in real time, for the maximum bitwidth (b_{max}) over the entire output. Pass 2 computes the optimal shift value based on the finalized b_{max}, then rereads the data and applies Shift-and-Round and Saturation to generate the final INT8 output.

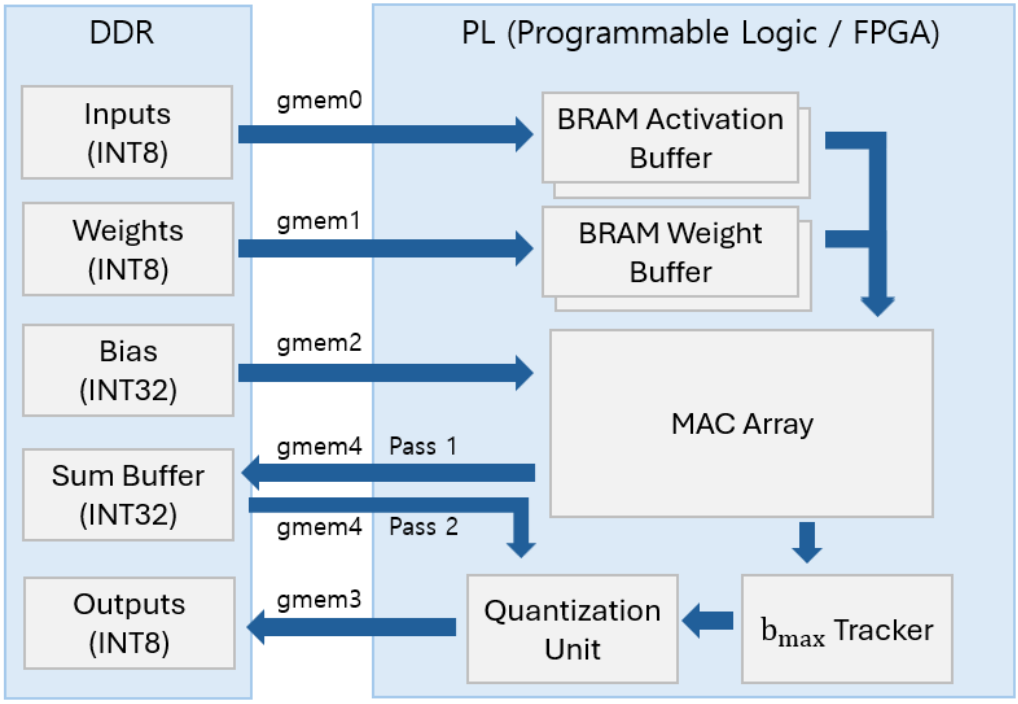

Figure 3. Block diagram of the proposed accelerator architecture featuring on-chip prefetching, multi-AXI parallel memory access, and a two-pass mechanism for integer-only dynamic scaling.

Currently, each tile is prefetched into BRAM before computation; we do not overlap prefetch and compute via a ping-pong scheme across tiles. Accordingly, the reported latency includes tile prefetch, on-chip MAC execution, and the extra sum_buf write/read overhead introduced by the two-pass scaling procedure.

3.6. SoC FPGA-Based Inference Parallelization Method #3: PS-Driven Orchestration for End-to-End Acceleration

The third strategy targets not only kernel throughput but also the reduction in system overhead that accumulates in end-to-end inference with repeated layer invocations. In a SoC FPGA environment, control overheads such as buffer allocation/deallocation, data copying, cache synchronization, register configuration and completion polling can accumulate and dominate the overall latency.

- Weight/Bias caching: The weights and biases of each layer are loaded only once into physically contiguous buffers and then reused across repeated inferences, thereby reducing re-allocation and repeated transfer overhead.
- Buffer Reuse and Pointer Swapping: IN/OUT/accumulation (SUM) work buffers sized for the maximum feature-map footprint are pre-allocated and reused. For layer segments without pooling, as illustrated in the execution loop of Figure 4, the next-layer input is formed by swapping buffer pointers rather than performing an actual data copy (the pooling result is written back to the IN buffer only when necessary).

After the PL-side convolution, pooling is handled by the PS and is executed in the INT8 domain without any floating-point conversion. The final FC/head is also processed using INT8 $\times$ INT8 $\rightarrow$ INT32 accumulation with the same RN-and-saturation rule, ensuring that the entire pipeline—from input to the final decision (i.e., argmax for classification and next-hop selection for routing)—is maintained as an integer-only path.

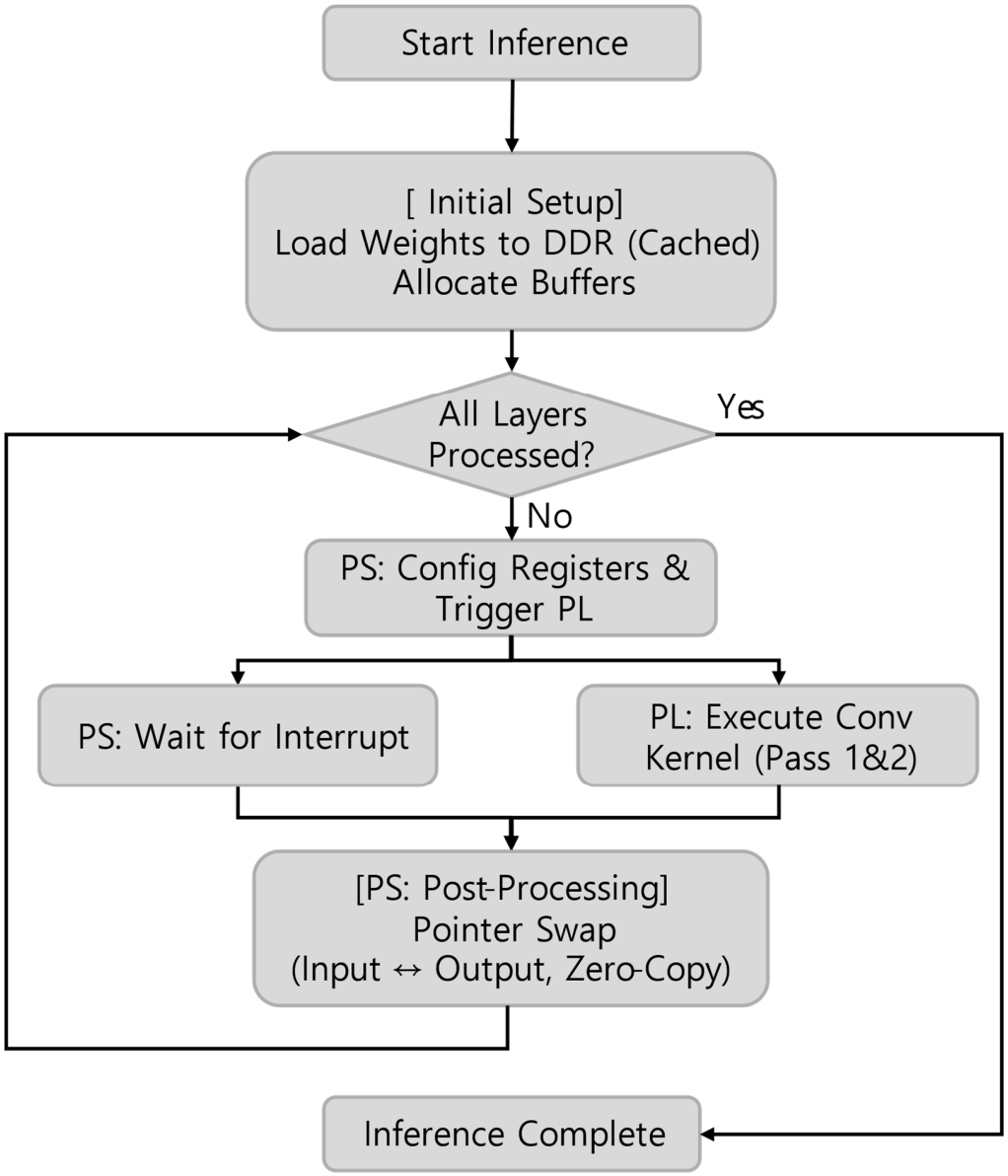

Figure 4. Flowchart of the proposed PS-driven inference orchestration.

4. Experimental Results

4.1. Experimental Setups

This section presents the experimental validation of the proposed PS–PL cooperative integer-only inference framework. We first describe the hardware and software setup, followed by a detailed analysis of functional correctness, end-to-end performance, layer-wise bottlenecks, and FPGA resource utilization. The experimental results demonstrate that the proposed approach achieves substantial acceleration while maintaining inference accuracy under strict onboard resource constraints.

4.1.1. Hardware/Software Environment and Implementation Parameters

Experiments were conducted on an AMD ZCU104 evaluation board equipped with a Zynq UltraScale+ MPSoC (XCZU7EV) (AMD, Santa Clara, CA, USA). On the PS side, the ARM Cortex-A53 quad-core processor (ARM Holdings, Cambridge, UK) executed inference orchestration and pre-/post-processing operations under PYNQ OS 2.7 (AMD, San Jose, CA, USA) with Python 3.8.2 and PyTorch 1.13.1. On the PL side, the convolution

accelerator IP was synthesized using Vitis HLS 2020.2 and integrated into the system design through Vivado 2020.2, then loaded as a PYNQ Overlay for runtime reconfiguration.

The PS–PL interface was configured with AXI4-Lite (ARM Holdings, Cambridge, UK) for control-path communication (register access and status polling) and AXI4 Memory-Mapped (ARM Holdings, Cambridge, UK) for data-path transfers. To maximize DDR bandwidth utilization, five independent AXI master ports were allocated: one each for input feature maps (in_r), weights (weights), biases (bias), output feature maps (out_r), and the INT32 accumulation buffer (sum_buf) required by the Two-Pass mechanism. Physical memory allocation on the PS was performed using PYNQ's allocate() API, and the corresponding physical addresses were written to IP control registers, enabling direct PL-to-DDR access without explicit data copying.

Beyond the platform description, the implemented kernel uses a maximum supported input channel bound of 512, and the actual PL clock frequency after routing was 96.97 MHz (with a positive timing slack of 0.408 ns). Model weights and biases were loaded once into cacheable physically contiguous buffers and reused across repeated inferences to minimize overhead.

4.1.2. Model and Integer-Only Inference Pipeline

The validation model is an INT8 VGG-SMALL-7-family CNN trained under the NITI framework, evaluated on the CIFAR-10 test set (32 $\times$ 32 RGB images, 10,000 samples). This model serves as a structural proxy for the Dueling-DQN feature extractor to strictly validate the hardware arithmetic fidelity. Input activations and weights are quantized to INT8, while intermediate MAC accumulation is performed in INT32 to prevent overflow. The numerical representation follows NITI's block exponentiation scaling scheme, which maintains per-layer scale exponents (s_a, s_w, s_{out}) throughout the forward pass.

To handle dynamic-range variations in convolution outputs within the integer domain, the PL accelerator employs a Two-Pass execution model. In Pass 1, INT32 accumulation results are written to accumulation buffer (sum_buf) while simultaneously tracking the maximum effective bitwidth (b_{max}) across all output elements. In Pass 2, a shift value is computed from b_max, and the final INT8 output is generated by applying shift-and-round (Round-to-Nearest) and saturation clipping. The PL then returns the output scale exponent ($s_{a,out}$) to the PS, which propagates it to the next layer as the input scale. Non-convolution operations—including ReLU activation, MaxPool (2 $\times$ 2, stride 2), and the final fully connected layer—are executed on the PS using the same INT8 arithmetic and shift-and-round rules.

4.1.3. Performance Metrics and Baseline Comparison

Performance evaluation focused on four key aspects: classification accuracy (Top-1 on CIFAR-10), end-to-end inference latency and throughput, per-layer execution time breakdown for convolution operations, and post-implementation FPGA resource utilization (LUT, FF, DSP, BRAM). Unless otherwise noted, the reported end-to-end latency refers to steady-state repeated inference after overlay loading and one-time buffer/model initialization. It includes per-layer register programming, cache-coherence operations (flush/invalidate), accelerator launch and completion polling, PS-side execution of ReLU/MaxPool/FC, and buffer reuse overhead. In contrast, the per-convolution timing used for layer-wise analysis is measured from ap_start issuance to accelerator completion, including completion polling and post-run invalidation, but excluding pre-launch register writes and cache flushes.

For the two-pass overhead profiling, each convolution layer was measured on the ZCU104 in three modes: Mode 0 (full two-pass execution), Mode 1 (first pass with INT32

sum-buffer materialization), and Mode 2 (first pass without full sum-buffer materialization). The latency decomposition was computed as:

$$T^{\text{est}}_{\text{pass2}} = T_{\text{full}} - T_{\text{pass1,store}}$$

$$T^{\text{est}}_{\text{sumwrite}} = T_{\text{pass1,store}} - T_{\text{pass1,nostore}}$$

where T_{full} is the latency of the complete two-pass execution (Mode 0), $T_{\text{pass1,store}}$ is the latency of the first pass including the INT32 intermediate-buffer write (Mode 1), and $T_{\text{pass1,nostore}}$ is the latency of the first pass without the full buffer materialization (Mode 2). By calculating the differences, we can isolate the effective time spent on the second sweep ($T^{\text{est}}_{\text{pass2}}$) and the off-chip memory write overhead of the intermediate sums ($T^{\text{est}}_{\text{sumwrite}}$).

The proposed system was compared against the following baseline:

- PS+PL (proposed): Convolution layers are offloaded to the PL accelerator, while ReLU, MaxPool, FC, and inference control logic are executed on the PS.
- PS-only baseline: All operations, including convolutions, are executed on the PS using the same INT8/INT32 arithmetic rules without PL offloading. This path preserves the same integer-only arithmetic rules used by the proposed implementation and is used here as a software reference baseline for functional and system-level comparison. It should therefore be interpreted as a PS-only software reference rather than as a fully optimized ARM INT8 inference engine.

4.2. Results and Analysis

4.2.1. Classification Accuracy

To confirm functional correctness and numerical stability of the integer-only pipeline, inference was performed on the full CIFAR-10 test set. On the full dataset evaluation, the PS+PL implementation achieved a Top-1 accuracy of 91.0%, while the PS-only baseline recorded 90.5%. These preliminary full-set results broadly confirm that the PL accelerator preserves the model's overall classification performance. Furthermore, to rigorously verify bit-level consistency between the PS-only and PS+PL paths, ensuring that the preliminary accuracy gap is merely statistical noise from unaligned evaluation sets, we conducted a controlled matched-subset evaluation. By feeding an identical fixed subset of input samples to both execution paths, the two implementations showed 100% sample-wise prediction agreement with zero mismatched predictions. This explicitly confirms that the PL accelerator correctly reproduces the intended integer-only arithmetic, including dynamic scaling and signed requantization, without any implementation-level mismatch.

The measured accuracy also closely matches the training-time validation accuracy (approximately 91%), demonstrating that the proposed Two-Pass dynamic scaling mechanism correctly reproduces NITI's quantization scheme in hardware without degradation.

4.2.2. End-to-End Inference Latency and Throughput

Table 4 compares the end-to-end performance of PS-only and PS+PL. Throughput is defined as $1/\text{latency}(\text{s})$. The speedup of PS+PL over PS-only is then calculated as the ratio of their respective mean latencies.

Table 4. End-to-End Latency and Throughput (PS-only vs. PS+PL).

Method	Mean Latency (s/Sample)	Std. Dev. (s)	Throughput (Samples/s)	Accuracy (%)	Speedup
PS-only	43.1169	0.151	0.0232	90.5%	1.00
PS+PL	2.9283	0.001	0.3415	91.0%	14.72

The proposed PS+PL system reduced the average end-to-end inference time from 43.117 s/sample in the PS-only software reference to 2.928 s/sample on the same ZCU104 platform, corresponding to a 14.72× speedup over the software reference path. The reported speedup is measured against our PS-only reference implementation on the same platform. It is not intended as a benchmark against a fully optimized ARM INT8 inference engine. These results highlight that offloading the integer-only convolution tasks to the PL provides a significant performance boost for the reference workflow within our target SoC environment. The PS+PL implementation achieves a mean latency of 2.928 s per CIFAR-10 image, representing a 14.72× speedup over the PS-only baseline. Correspondingly, throughput improves from 0.023 to 0.342 images/s. This confirms that offloading the convolution-dominant feature extraction to the PL substantially reduces inference time on the target SoC FPGA. It should be noted that processing a single image in this proxy workload is not strictly equivalent to a complete next-hop decision in a Grid-MDP routing loop. Accordingly, Table 4 should be interpreted as evidence of the hardware feasibility of the proposed PS–PL pipeline, not as a direct latency estimate for the complete routing loop. The measured latency reduction nevertheless shows that the proposed architecture effectively mitigates the dominant computational bottleneck in the reference workflow.

4.2.3. Layer-Wise Convolution Performance and Bottleneck Analysis

To understand the source of the end-to-end speedup and identify remaining bottlenecks, Figure 5 presents a breakdown of mean convolution execution time for each layer (Conv1 through Conv6), along with per-layer speedup factors.

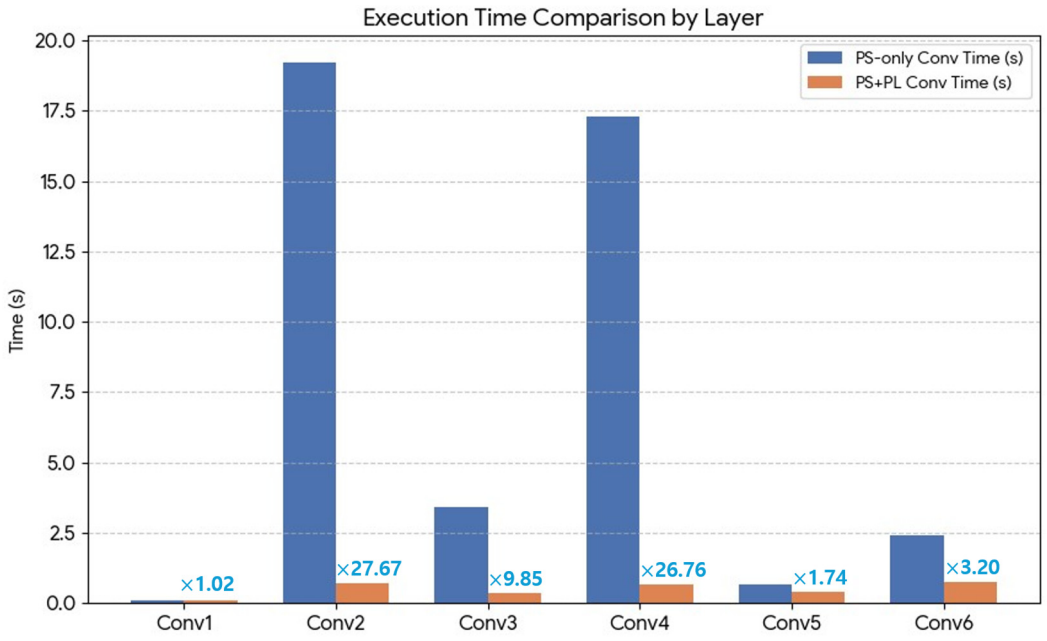

Figure 5. Mean Convolution Time per Layer and Speedup (PS-only vs. PS+PL). The reported times correspond to the end-to-end on-board accelerator invocation, inclusive of PS–PL synchronization and memory coherence overhead. Mean per-convolution accelerator-call time measured from ap_start to completion, including completion polling and output-buffer invalidation, but excluding pre-launch register writes and cache flushes.

The layer-wise analysis reveals that the computational bottleneck in the PS-only baseline is primarily concentrated in Conv2 and Conv4, which account for the majority of the total convolution time. PL offloading addresses this by delivering substantial acceleration for these compute-intensive layers—specifically achieving 27.67× speedup for Conv2 and 26.76× for Conv4—where the workload is sufficient to amortize invocation overhead

and fully utilize the parallel MAC array. These gains directly translate to the observed end-to-end speedup, confirming the hypothesis that convolution operations constitute the dominant bottleneck in the CNN-based DRL feature extractor. Furthermore, the non-uniform layer-wise speedup profile highlights opportunities for further optimization, such as reducing fixed overheads for smaller layers and exploring layer fusion or streaming execution models.

4.2.4. Quantitative Overhead Breakdown of the Two-Pass Design

Table 5 and Figure 6 quantify the overhead of the proposed two-pass mechanism. The exact additional external-memory traffic is 1,048,576 bytes for Conv1–2, 524,288 bytes for Conv3–4, and 262,144 bytes for Conv5–6, which corresponds to the analytical expression $8 \cdot C_{out} \cdot H_{out} \cdot W_{out}$.

Table 5. Layer-wise overhead decomposition of the two-pass mechanism.

Layer	Extra DDR Traffic (Bytes)	Estimated Sum-Buffer Materialization Latency (ms)	Estimated Second-Sweep Latency (ms)	Total Layer Time (ms)
Conv1	1,048,576	46.187	3.952	84.515
Conv2	1,048,576	46.180	4.069	693.400
Conv3	524,288	23.031	3.601	342.427
Conv4	524,288	23.098	3.486	646.506
Conv5	262,144	11.556	3.225	388.815
Conv6	262,144	11.567	3.214	753.720

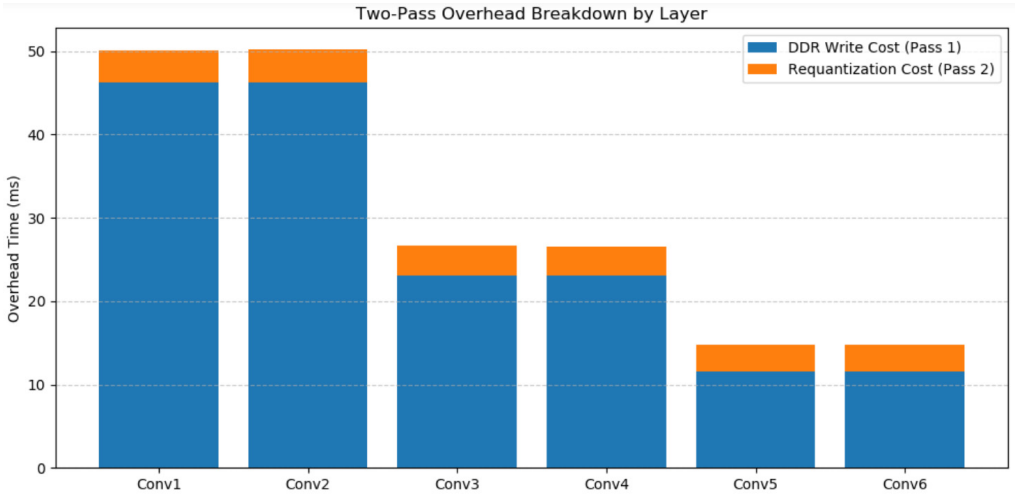

Figure 6. Layer-wise overhead decomposition of the two-pass mechanism.

The estimated latency decomposition shows that the dominant overhead comes from the INT32 intermediate-buffer materialization ($T^{\text{est}}_{\text{sumwrite}}$), whose cost decreases almost proportionally with the output tensor size. In contrast, the estimated second-sweep latency ($T^{\text{est}}_{\text{pass2}}$) remains comparatively small, ranging from 3.21 ms to 4.07 ms across all layers. These results indicate that the main penalty of the two-pass design arises from the off-chip materialization of the intermediate INT32 sums rather than the requantization arithmetic itself.

More importantly, for the compute-intensive layers that dominate the total execution time (e.g., Conv2, Conv4, Conv6), the relative overhead of the two-pass mechanism becomes marginal (less than 7%).

4.2.5. Ablation of Tunable Design Choices

To quantify the contributions of design choices that can be varied without changing the architectural validity of the accelerator, we consider a 2×2 ablation over PE array size (8×8 vs. 16×16) and PS-side buffer reuse (off vs. on), while keeping on-chip prefetching and output-stationary scheduling fixed across all variants. On-chip prefetching is kept fixed because, under the fully unrolled PE array used in our design, removing it would invalidate the intended memory hierarchy and lead to an unfair comparison point rather than a meaningful optimization ablation.

As shown in Table 6, increasing the PE array size from 8×8 to 16×16 substantially reduces the PL kernel latency, indicating that compute scaling is a dominant contributor to the performance improvement (yielding a 1.69× speedup at the no-reuse baseline).

Table 6. Ablation of tunable design choices in the PS–PL cooperative accelerator.

Variant	PE Array	Buffer Reuse	End-to-End Latency (s)	PS Overhead (s)	PL Kernel (s)	Speedup
A0	8×8	Off	5.654295	0.460653	5.187112	1.00×
A1	8×8	On	5.192637	0.000015	5.186230	1.09×
A2	16×16	Off	3.343624	0.414961	2.922115	1.69×
A3 (Proposed)	16×16	On	2.927101	0.000015	2.920867	1.93×

Furthermore, as shown in Table 6, PS-side buffer reuse produces little change in the PL kernel time, but consistently reduces the end-to-end latency by removing repeated host-side allocation, copy, and deallocation overhead. As indicated by the speedups relative to A0, the incremental gain of buffer reuse increases from 1.09× (at 8×8, A0 to A1) to 1.14× (at 16×16, A2 to A3), demonstrating that host-side orchestration optimization becomes more critical as the PL kernel execution becomes faster. Combined, these independently tunable axes reduce the end-to-end latency by 1.93× compared to the reduced-parallelism reference design (A0).

4.2.6. FPGA Resource Utilization

Table 7 summarizes the post-implementation resource utilization for the proposed PS–PL accelerator as reported by Vivado after place-and-route on the ZCU104 device (XCZU7EV).

Table 7. Post-Implementation FPGA Resource Utilization of the Proposed PS–PL Accelerator.

Resource	Used	Available	Utilization (%)
LUT	134,080	230,400	58.19
LUTRAM	3235	101,760	3.18
FF	98,955	460,800	21.47
BRAM	155	312	49.68
DSP	559	1728	32.35

The design consumes 58.19% of available LUTs, 49.68% of BRAMs, and 32.35% of DSP slices, while FF utilization remains relatively low at 21.47% and LUTRAM at only 3.18%. This resource distribution reflects the architectural trade-offs inherent in the proposed design.

BRAM utilization (49.68%) is primarily driven by on-chip tile buffers required for the prefetching strategy described in Section 3.5. By buffering weight and input tiles in BRAM, the design eliminates DDR accesses in the innermost accumulation loop, significantly reducing memory traffic and improving pipeline efficiency. The cost of this optimization is an increased on-chip memory footprint, particularly because buffers are sized to accommodate worst-case layer dimensions rather than being dynamically allocated per layer.

LUT utilization (58.19%) arises from address generation and indexing logic for output-stationary tiling, control flow for tile-boundary handling and padding, and additional combinational logic introduced by loop unrolling and pipelining directives in the HLS implementation. The parallel MAC array and multi-AXI memory interface also contribute to increased routing complexity.

The moderate DSP utilization (32.35%) indicates that the design is not compute-bound. This leaves sufficient headroom to further increase the parallelism factors (SA_M, SA_N) in the MAC array, thereby potentially achieving higher throughput in future iterations. Conversely, the low utilization of FF (21.47%) and LUTRAM (3.18%) indicates that the design is not constrained by sequential logic or small-scale storage.

From a scalability perspective, LUT and BRAM are the primary limiting factors. Future optimizations should focus on adapting buffer sizes to per-layer channel counts rather than using fixed worst-case allocations and exploring alternative buffer partitioning schemes to reduce BRAM consumption. Overall, the resource utilization profile demonstrates that the proposed architecture fits comfortably within the constraints of a mid-range SoC FPGA (ZCU104), providing a feasible solution for onboard satellite environments.

5. Conclusions

This paper presented a PS–PL cooperative integer-only inference framework to accelerate convolution-dominant CNN inference on a heterogeneous SoC FPGA for onboard LEO satellite routing. The core contribution of the proposed design lies in the integration of NITI-style two-pass scaling with PS–PL exponent propagation, preserving end-to-end integer consistency without floating-point conversion. Through output-stationary tiling, on-chip prefetching, and PS-side buffer reuse, the implementation reduces memory traffic and invocation overhead. On a ZCU104 platform, the proposed PS–PL scheme achieved a 14.7× end-to-end speedup over a PS-only integer baseline while maintaining comparable accuracy on an INT8 VGG-style model. Furthermore, an ablation study over independently tunable axes demonstrates that combining PL-side compute scaling with PS-side buffer reuse jointly yields a 1.93× latency reduction over a reduced-parallelism reference. These results demonstrate the hardware feasibility of a low-latency integer-only CNN inference prototype on a SoC FPGA under tight onboard resource constraints. While the proposed PS–PL framework substantially reduces proxy-workload latency, validating real-time suitability with the actual routing CNN in a complete Grid-MDP routing loop remains future work.

Author Contributions: Conceptualization, B.J. and H.L.; Methodology, B.J.; Software, B.J.; Validation, B.J.; Formal analysis, B.J.; Investigation, B.J.; Resources, H.L.; Data curation, B.J.; Writing–original draft, B.J.; Writing–review & editing, B.J., H.L., B.R. and M.H.; Visualization, B.J.; Supervision, H.L.; Project administration, H.L., B.R. and M.H.; Funding acquisition, B.R. and M.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Agency for Defense Development, funded by the Korean government (Defense Acquisition Program Administration) (UI247034TF).

Data Availability Statement: The original data is not available due to the policy of the funder.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Al-Hraishawi, H.; Chougrani, H.; Kisseleff, S.; Lagunas, E.; Wang, S.; Ottersten, B. A Survey on Nongeostationary Satellite Systems: The Communication Perspective. *IEEE Commun. Surv. Tutor.* **2023**, *25*, 101–132. [CrossRef]
2. Westphal, C.; Choudhury, A.; Raychaudhuri, D.; Benmammar, B.; Birrane, E.; Boswell, S. LEO Satellite Networking Relaunched: Survey and Current Research Challenges. *ITU J. Future Evol. Technol.* **2023**, *4*, 711–744. [CrossRef]

3. Handley, M. Delay is Not an Option: Low Latency Routing in Space. In Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18), Redmond, WA, USA, 15–16 November 2018; pp. 85–91. [CrossRef]
4. Wang, J.; Li, L.; Zhou, M. Topological Dynamics Characterization for LEO Satellite Networks. *Comput. Netw.* **2007**, *51*, 43–53. [CrossRef]
5. Cao, X.; Wang, Y.; Yang, F.; Gao, J. Dynamic Routings in Satellite Networks: An Overview. *Sensors* **2022**, *22*, 4552. [CrossRef] [PubMed]
6. Ekici, E.; Akyildiz, I.F.; Bender, M.D. A Distributed Routing Algorithm for Datagram Traffic in LEO Satellite Networks. *IEEE/ACM Trans. Netw.* **2001**, *9*, 137–147. [CrossRef]
7. Akyildiz, I.F.; Ekici, E.; Bender, M.D. MLSR: A Novel Routing Algorithm for Multilayered Satellite IP Networks. *IEEE/ACM Trans. Netw.* **2002**, *10*, 411–424. [CrossRef]
8. Lu, Y.; Sun, F.; Zhao, Y. Virtual Topology for LEO Satellite Networks Based on Earth-Fixed Footprint Mode. *IEEE Commun. Lett.* **2013**, *17*, 357–360. [CrossRef]
9. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
10. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-Level Control through Deep Reinforcement Learning. *Nature* **2015**, *518*, 529–533. [CrossRef] [PubMed]
11. Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; de Freitas, N. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016), New York, NY, USA, 20–22 June 2016*; Balcan, M.F., Weinberger, K.Q., Eds.; Proceedings of Machine Learning Research; PMLR: New York, NY, USA, 2016; Volume 48, pp. 1995–2003.
12. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
13. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2014**, arXiv:1409.1556.
14. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15), Monterey, CA, USA, 22–24 February 2015; pp. 161–170. [CrossRef]
15. Sharma, H.; Park, J.; Suda, N.; Lai, L.; Chauhan, M.; Chandra, V.; Esmaeilzadeh, H. From High-Level Deep Neural Models to FPGAs. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49), Taipei, China, 15–19 October 2016; pp. 1–12. [CrossRef]
16. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–22 June 2018; pp. 2704–2713. [CrossRef]
17. Wang, P.; Lin, Q.; Liu, X.; Ma, J.; He, X.; Hu, J. NITI: Training Integer Neural Networks Using Integer-Only Arithmetic. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 3249–3261. [CrossRef]
18. Krizhevsky, A.; Hinton, G. *Learning Multiple Layers of Features from Tiny Images*; Technical Report; University of Toronto: Toronto, ON, Canada, 2009.
19. Yu, C.; Kim, D.; Lee, H.; Han, M. GPU-Accelerated CNN Inference for Onboard DQN-Based Routing in Dynamic LEO Satellite Networks. *Aerospace* **2024**, *11*, 1028. [CrossRef]